

# **LAB 5 REPORT**

**Course: CPS633**

**Section: 6**

## **Group Members:**

Alishba Aamir, 500974648

Ivan Golovine, 500813431

Fangbo Ren, 500884730

## Task 1: Generate Encryption Key in a Wrong Way

With srand (time(NULL));

```
File Machine View Input Devices Help
Terminal
[10/17/21]seed@VM:~$ cd Labs
[10/17/21]seed@VM:~/Labs$ cd lab5
[10/17/21]seed@VM:~/.../lab5$ ls
task1.c
[10/17/21]seed@VM:~/.../lab5$ gcc task1.c
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634484902
a76d1d7d05225d1589c42d04b7406090
[10/17/21]seed@VM:~/.../lab5$
```

Without srand (time(NULL));

```
Terminal
[10/17/21]seed@VM:~$ cd Labs
[10/17/21]seed@VM:~/Labs$ cd lab5
[10/17/21]seed@VM:~/.../lab5$ ls
task1.c
[10/17/21]seed@VM:~/.../lab5$ gcc task1.c
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634484902
a76d1d7d05225d1589c42d04b7406090
[10/17/21]seed@VM:~/.../lab5$ vim task1.c
[10/17/21]seed@VM:~/.../lab5$ gcc task1.c
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485017
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$
```

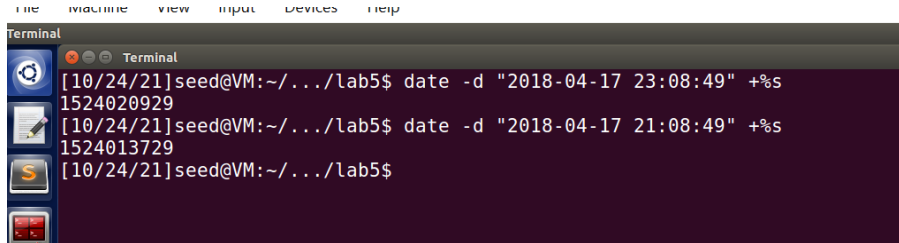
9bd2bce5098d2d27e89b1be43976bcc  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485467  
8f2f8917640a42e7a80632ddacfe83c4  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485468  
2c25dc2e90fae4b44f87931decae83bc  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485469  
01eab2f8b6a7e0cfc389553954e5eb6e  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485470  
4616e60e55cb3ef1fe2c8020091010a  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485471  
fe1ec6a71e88d5e8c647b88ded9aef52  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485472  
84e3f342075a6b4bdc7f56870c3dc20f  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485472  
84e3f342075a6b4bdc7f56870c3dc20f  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485473  
5f69dfe180a08e5d65c74220c3e7b0bb  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485474  
e15a7d68a803253884253ec5cb1b5739  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out  
1634485474  
e15a7d68a803253884253ec5cb1b5739  
[10/17/21]seed@VM:~/.../lab5\$ ./a.out

```
Terminal
[10/17/21]seed@VM:~/.../lab5$ vim task1.c
[10/17/21]seed@VM:~/.../lab5$ gcc task1.c
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485611
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485612
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485613
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485613
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485614
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485615
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485616
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$ ./a.out
1634485617
67c6697351ff4aec29cdbaabf2f346
[10/17/21]seed@VM:~/.../lab5$
```

The purpose of the time() function is to return the current calendar time. If you run the prog **with** the srand (time(NULL)) a couple of times, you will notice the first line is the current time while the second line is the encrypted key. The srand(time(NULL)) function uses the computer clock(current calendar time) as the seed. Since time is always changing, the seed will also be changing. This means every time, you run the program, the encrypted key will be different. You can see the evidence by comparing the 2 outputs above. For the program running **without** the srand(time(NULL)) function, the time is different **BUT** the encrypted key is the **SAME** in all the runs. For the program running **with** the srand(time(NULL)) function, the encrypted key is the **DIFFERENT** With every run.

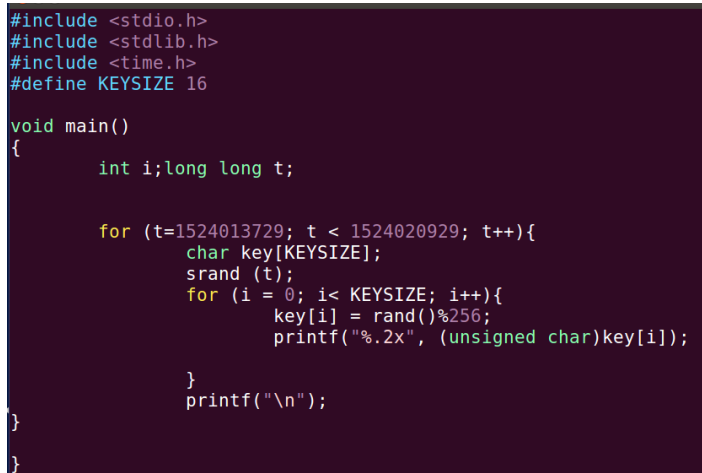
## TASK 2: Guessing the Key

First, we are going to use the date command to find time in second for range of 2 hours for the timestamp that we were given. Therefore, the file could have been encrypted anywhere between 1524013729 to 1524020929.

A terminal window titled 'Terminal' with a menu bar (File, Machine, View, Input, Devices, Help). The prompt is [10/24/21]seed@VM:~/.../lab5\$. The user enters 'date -d "2018-04-17 23:08:49" +%s' and the output is 1524020929. Then the user enters 'date -d "2018-04-17 21:08:49" +%s' and the output is 1524013729. The prompt returns to [10/24/21]seed@VM:~/.../lab5\$.

```
Terminal
[10/24/21]seed@VM:~/.../lab5$ date -d "2018-04-17 23:08:49" +%s
1524020929
[10/24/21]seed@VM:~/.../lab5$ date -d "2018-04-17 21:08:49" +%s
1524013729
[10/24/21]seed@VM:~/.../lab5$
```

Now, we used the c program given to us in Task 1 to generate the list of possible keys that could have been used to encrypt the document. Here, instead of using the current time, we incremented the time one by one starting from the first possible time, the key could have been made to the last possible time the key could have been made. We run this program and put the list of possible keys in key.txt.

A C program code snippet shown in a terminal window. It includes headers <stdio.h>, <stdlib.h>, and <time.h>. It defines KEYSIZE as 16. The main function starts with a loop for t from 1524013729 to 1524020929. Inside the loop, it declares a char array key of size KEYSIZE, calls srand(t), and then loops for i from 0 to KEYSIZE-1, assigning key[i] = rand()%256 and printing it in hexadecimal format. After the inner loop, it prints a newline. The program ends with a closing brace for main and a final closing brace for the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
    int i; long long t;

    for (t=1524013729; t < 1524020929; t++){
        char key[KEYSIZE];
        srand (t);
        for (i = 0; i < KEYSIZE; i++){
            key[i] = rand()%256;
            printf("%.2x", (unsigned char)key[i]);

        }
        printf("\n");
    }
}
```

Next, with the help of a python program, we took each key from the list of keys and used it along with the iv, and data to figure out the guess key. If our key and cipher text match, then that means we have successfully guessed the correct key used to encrypt the document.

```
Terminal
from Crypto.Cipher import AES

data = bytearray.fromhex('255044462d312e350a25d0d4c5d80a34')
ciphertext = bytearray.fromhex('d06bf9d0dab8e8ef880660d2af65aa82')
iv = bytearray.fromhex('09080706050403020100A2B2C2D2E2F2')

with open("./key.txt") as f:
    keys = f.readlines()

for k in keys:
    k = k.rstrip('\n')

    key = bytearray.fromhex(k)

    cipher = AES.new(key=key, mode=AES.MODE_CBC,iv=iv)

    guess = cipher.encrypt(data)

    if guess == ciphertext:
        print("THE KEY IS: ",k)
```

As seen below, we run our task2.c prog to generate the list of possible keys. Then we run our generate.py program which uses the list of keys to guess a key.

Our guess for the key is : 95fa2030e73ed3f8da761b4eb805dfd7

```
Terminal
[10/25/21]seed@VM:~/.../lab5$ gcc task2.c -o task2
[10/25/21]seed@VM:~/.../lab5$ ./task2 > key.txt
[10/25/21]seed@VM:~/.../lab5$ vim key.txt
[10/25/21]seed@VM:~/.../lab5$ vim generate.py
[10/25/21]seed@VM:~/.../lab5$ python3 generate.py
THE KEY IS: 95fa2030e73ed3f8da761b4eb805dfd7
[10/25/21]seed@VM:~/.../lab5$
```

### Task 3: Measure the Entropy of Kernel

#### Initial Entropy

```
[10/24/21]seed@VM:~/.../lab5$ cat /proc/sys/kernel/random/entropy_avail  
394  
[10/24/21]seed@VM:~/.../lab5$
```

Moving the mouse would generally produce a slow consistent amount of entropy, when doing it in addition to other things it was effective but on its own it was descent for consistent production.

```
Terminal  
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail  Sun Oct 24 17:56:48 2021  
883  
  
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail  Sun Oct 24 22:12:18 2021  
893
```

Typing would be faster at building up the entropy but if there isn't a large amount of entropy built up would have periods of time where it would also jump lower.

-----  
Reading a large file would usually result in a big jump in entropy of 100 or more up or down, especially when combined with typing and mouse movement.

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail  Sun Oct 24 18:13:11 2021  
3327  
  
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail  Sun Oct 24 18:17:19 2021  
3549
```

Visiting a website would result in similar jump up or down to opening a large file.

### Task 4: Get Pseudo Random Numbers from /dev/random

When you don't move the mouse the number increases by 1 every second and then dumps in hexcode when it reaches the 60's in randomness.



## Task 5: Get Random Numbers from /dev/urandom

When running `cat/dev/urandom | hexdump`, it keeps printing out the random numbers.

```
071c0b0 6f41 a33c 76af da52 9a57 d3d1 a58b bd1b
071c0c0 c943 6ded e506 eff0 dc87 552e a537 b55c
071c0d0 f2a5 6ac8 5926 dc3c baf2 682a b3b4 fb35
071c0e0 315b bfea e27b de49 2b63 09bf 9d88 7a61
071c0f0 db5f d783 c135 890a cf71 5e15 1d8a 5f14
071c100 0468 eefd b2ad 5a24 2fc8 d051 b80a 8843
071c110 5e5e c88e cc43 1d26 44cd 3c15 2ef7 3eec
071c120 de88 98f0 2543 3f3f 435d 89f8 a52d 89c5
071c130 9e56 3bd4 ee40 90ac 0828 59ec 4550 0ba6
071c140 63da 88e6 1c9b d4c5 a1af 667b 24fc 7074
071c150 1755 c4ac 32c4 e2b1 be30 d5f0 f9df 28ab
071c160 4141 a893 f91a eab7 b835 2681 112f 60eb
071c170 dd57 3cff 4895 2164 7379 a3e3 4bc1 208b
071c180 31b2 3079 25b8 ff55 df28 f2e6 0979 4ccd
071c190 40c6 41bd e5ae 4ce5 f1ca 25ab 55ee 1579
071c1a0 45ce c776 2cc5 1e93 96ef 46ea f911 9e0c
071c1b0 c453 543b d442 b745 4fa8 b4bd 3a52 cfad
071c1c0 a85b c968 98df 8ba0 07fe 8619 2759 2f20
071c1d0 326d c877 c25b eb95 4b39 1e02 8f6a 74a4
071c1e0 1973 2707 eda4 387f 44e6 9ca0 3ef6 5d3d
071c1f0 db1c 32d9 22ae cced 1e0d 804f 7d91 c84c
071c200 2be3 dad0 82a6 ebac e2af d8b2 e509 30e9
071c210 acdc b302 e234 6d5d 2691 1567 f83a cfdb
071c220 c9cd 51e4 75ca 9c9f d6fc 359c 87f9 f387
```

The bellow image shows I truncate the first 1 MB outputs into a file named output.bin and Then use `ent` to evaluate its information density.

```
[10/24/21]seed@VM:~$ head -c 1M /dev/urandom > output.bin
[10/24/21]seed@VM:~$ ent output.bin
Entropy = 7.999802 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 287.19, and randomly
would exceed this value 8.10 percent of the times.

Arithmetic mean value of data bytes is 127.5748 (127.5 = random).
Monte Carlo value for Pi is 3.142147606 (error 0.02 percent).
Serial correlation coefficient is 0.000779 (totally uncorrelated =
0.0).
```

I modify the code, compile the `get_urandom.c` and run it to generate a 256-bit random number.



```
Terminal
GNU nano 2.5.3      File: get_urandom.c      Modified

#include <stdio.h>
#include <stdlib.h>
#define LEN 32 // 256 bits

int main()
{
    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned c$
    FILE *random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char) * LEN, 1, random);
    fclose(random);
    printf("k = ");
    for (int i = 0; i < LEN; i++)
        printf("%.2x", key[i]);
    printf("\n");
    return 0;
}
```

```
[10/24/21]seed@VM:~$ nano get_urandom.c
[10/24/21]seed@VM:~$ gcc get_urandom.c -o get_urandom
[10/24/21]seed@VM:~$ ./get_urandom
k = 385a7fc635bcb5bf0b06d0e882dcbd4d89c3bc8ddd64a64cc556a5557701383
e
[10/24/21]seed@VM:~$
```